Documentation NexaScrore

Produit : NexaScore

Entreprise : NexaScor

Version : 1.0.0

Date : Novembre 2025

Auteur : VIART Vivien & SOBKOWIAK Axel

📄 Sommaire

---

1. Présentation du Projet

NexaTalent est une solution logicielle B2B destinée à moderniser les processus de recrutement des ETI et Grandes Entreprises. Face à l'afflux de CVs non qualifiés, notre mission est simple : **Rationaliser la présélection.**

Notre produit phare, **NexaScore**, est un moteur de scoring déterministe qui remplace l'intuition par la data. Il analyse mathématiquement la compatibilité entre une offre d'emploi et une base de candidats, garantissant une objectivité totale et un gain de temps estimé à **70%** pour les équipes RH.

---

2. Guide Fonctionnel (Utilisation)

Ce guide décrit les fonctionnalités accessibles via l'interface web sécurisée.

2.1. Tableau de Bord Stratégique (Dashboard)

Dès la connexion, le recruteur accède à une vue d'ensemble de son activité :

- **Indicateurs Clés (KPIs)** : Suivi en temps réel du volume d'offres ouvertes et de la taille du vivier de talents
- **Data-Visualisation** :
  - *Répartition des Besoins* : Un graphique analyse les types de postes les plus recherchés par l'entreprise
  - *Analyse du Vivier* : Un diagramme en anneau montre la séniorité des candidats disponibles (Junior / Confirmé / Senior)
- **Live Feed** : Un flux d'activité permet de suivre les dernières actions de l'équipe (création de poste, entrée d'un nouveau profil)

2.2. Gestion du Vivier & des Offres (CRUD)

L'application offre une interface ergonomique pour gérer les données :

- **Fiches Offres** : Création détaillée incluant le titre, la description, la ville cible et les compétences techniques requises (avec niveau d'exigence de 1 à 5)
- **Fiches Candidats (Smart Profile)** : Vue type "Carte de visite" regroupant l'état civil, l'expérience (calculée dynamiquement) et les compétences acquises
- **Référentiel Compétences** : Gestion centralisée des tags (C#, Java, Gestion de projet...) pour standardiser les critères de recherche

2.3. Le Moteur NexaScore (Matching)

C'est la fonctionnalité "Core Business" de l'application.

1. **Lancement** : Depuis une offre, le recruteur clique sur "Lancer l'analyse"
2. **Calcul** : Le système compare l'offre sélectionnée avec *tous* les candidats de la base
3. **Résultat** : Un rapport d'audit est généré, classant les candidats par pertinence (Score sur 100)
4. **Transparence** : Pour chaque candidat, le système explique la note via des indicateurs :
   - *"Points forts : Même ville, Niveau Expert C#"*

- *"Points faibles : Manque d'expérience"*

---

3. Architecture Technique

Cette section détaille les choix d'ingénierie logicielle garantissant la robustesse de NexaTalent.

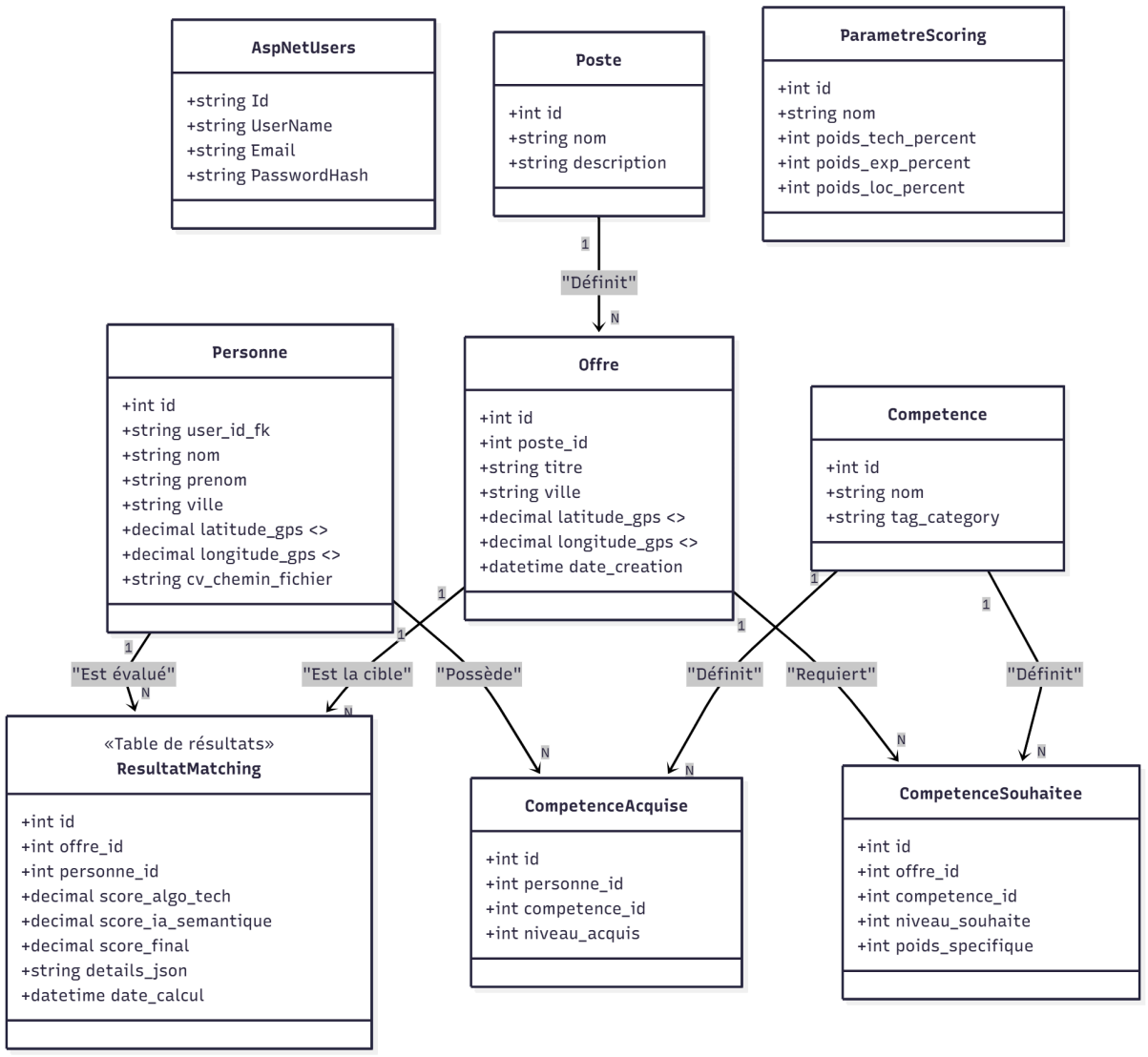
3.1. Stack Technologique

- **Backend** : C# (.NET Core 8.0) - Choisi pour sa performance et son typage fort
- **Architecture** : MVC (Model-View-Controller) strict
- **Base de Données** : SQL Server - Gestion relationnelle robuste
- **ORM** : Entity Framework Core - Pour la manipulation sécurisée des données (protection contre les injections SQL)
- **Frontend** : Razor, Bootstrap 5 (Customisé "NexaTheme"), Chart.js

3.2. Modèle de Données (Schéma SQL)

La base de données NexaTalent_DB est structurée autour de **7 tables clés** :

- **Entités Principales** : Offre, Personne (Candidat), Poste (Métier)
- **Référentiel** : Competence
- **Relations (Many-to-Many)** :
  - CompetenceAcquise : Relie Personne ↔ Compétence (avec attribut *Niveau*)
  - CompetenceSouhaitee : Relie Offre ↔ Compétence (avec attribut *NiveauRequis*)
- **Configuration** : ParametreScoring permet de définir les poids de l'algorithme par offre



3.3. Design & UX

L'interface utilisateur respecte les standards modernes du SaaS :

- **Glassmorphism** : Utilisation de transparences et de flous pour la barre de navigation
- **Feedback Utilisateur** : Jauges de progression, badges de couleur (Vert/Orange/Rouge) selon les scores
- **Responsive** : Accessible sur tablette et desktop

---

4. Logique Algorithmique — Le Cœur Hybride du Système

NexaScore repose sur une appro che **hybride** combinant :

- un **scoring déterministe** (Hard Skills)
- une **analyse sémantique IA** (Soft Skills & contexte)

Cette double analyse permet un score final à la fois **objectif** et **contextuel**.

4.1. Couche 1 : Le Scoring Déterministe (C#)

La première couche applique un calcul vectoriel pondéré exécuté dans le backend **.NET**.

Formule du score algorithmique

Pour chaque candidat, on calcule :

```
S_Algo = ((S_Tech × P_Tech) + (S_Exp × P_Exp) + (S_Loc × P_Loc)) / 100
```

Variables :

- **S_Tech (Score Technique)**

  Ratio exact entre compétences demandées et compétences possédées.

  Exemple : un niveau 3/5 = 60% des points.

- **S_Exp (Score Expérience)**

  Basé sur des paliers :

  - Junior → 40 pts
  - Confirmé → 70 pts
  - Senior → 100 pts

- **S_Loc (Score Localisation)**

  Binaire : **0 ou 100**

  Peut inclure un **Kill Switch** si le candidat est hors zone.

4.2. Couche 2 : Analyse Sémantique (API Python IA)

Après le score déterministe, le système envoie les données à un micro-service IA pour affiner le contexte.

Architecture

- Le backend C# transmet un fichier **JSON**

  → *Description du poste + CV parsé du candidat*
- L'API Python (FastAPI / Flask) traite la similarité sémantique.

Technologie

- Modèles NLP :
  - Transformers
  - TF-IDF
  - Similarité cosinus entre les textes

Détail de l'Interopérabilité (.NET / Python)
La communication entre le C# et le python est asynchrone pour ne pas ralentir l'expérience utilisateur :

1. **Serialization :** C# utilise `System.Text.Json` pour formater le payload.
2. **Transport :** Utilisation de `HttpClientFactory` pour gérer efficacement les sockets HTTP (évite le "socket exhaustion").
3. **Résilience :** Implémentation d'une politique de "Retry" (via la librairie **Polly**) : si l'IA ne répond pas en 2 secondes, le système réessaie 3 fois avant d'échouer proprement (Graceful Degradation).

Objectif

Détecter les **mots-clés contextuels**, synonymes et compétences proches.

Exemple : détecter que "ReactJS" et "VueJS" sont liés en contexte Front-End, ce qu'un SQL strict ne peut pas faire.

4.3. Score Final Unifié : NexaScore Global

Le score final présenté au recruteur est un mix pondéré des 2 couches :

```
S_Final = (S_Algo × 0.7) + (S_IA × 0.3)
```

Note

Dans cet exemple :

- **70%** = logique rationnelle
- **30%** = analyse IA (insights contextuels)

---

5. Installation et Déploiement

Prérequis

- Environnement Windows / Visual Studio 2022
- SQL Server LocalDB installé

Procédure de Démarrage

1. Cloner le dépôt du projet NexaTalent
2. Vérifier la chaîne de connexion `DefaultConnection` dans `appsettings.json`
3. Initialiser la base de données :
    - Ouvrir la console NuGet
    - Lancer la commande : `Update-Database` (ou exécuter le script SQL fourni)
4. Lancer l'application via IIS Express (Touche F5)

---

6. Évolutions Majeures

Cette version introduit une refonte complète de l'expérience utilisateur pour passer d'un simple "CRUD administratif" à une véritable application métier (SaaS).

6.1. Workflow de Création en Étapes (Wizard Pattern)

Problématique technique : Lors de la création d'un Candidat ou d'une Offre, il est impossible d'associer des compétences immédiatement car l'entité n'a pas encore d'ID en base de données (Primary Key non générée).

Solution implémentée : Nous avons développé un **Workflow Séquentiel (Wizard)** :

1. **Étape 1 (Création)** : La RH saisit les informations des personnes (Nom, Poste, Ville)
2. **Transition** : Le contrôleur sauvegarde, récupère l'ID généré, et redirige automatiquement vers l'édition avec un message de succès (TempData)

3. **Étape 2 (Enrichissement)** : L'utilisateur arrive sur l'interface de configuration pour ajouter les compétences et critères de scoring
4. **Visuel** : Une barre de progression guide l'utilisateur (✓ Étape 1 validée → Étape 2 active)

6.2. Intégration d'APIs Tiers (Géo-Intelligence)

Pour fiabiliser les données géographiques (important pour le calcul de score), nous avons connecté l'application à des services externes.

Autocomplétion (API Gouv)

- Sur les formulaires, le champ "Ville" interroge en temps réel l'API `geo.api.gouv.fr`
- **Bénéfice** : Évite les fautes de frappe (ex: "Lile" alors que c'est "Lille") et remplit automatiquement le Code Postal (ici 59000)
- **Technique** : JavaScript asynchrone (exécution sans bloquer le reste du programme) (fetch) + Manipulation de l'interface.

Cartographie Interactive (Leaflet.js + OpenStreetMap)

- Sur les fiches de détails, une carte interactive affiche la position exacte du poste ou du candidat
- **Géocodage** : Un script convertit l'adresse textuelle ("Lille, 59000") en coordonnées GPS (Lat/Lon) via l'API Nominatim pour placer le marqueur
- **Avantage** : Solution 100% Open Source et gratuite (contrairement à Google Maps)

6.3. Optimisation de la Productivité (Bulk Actions)

Pour éviter aux recruteurs des clics répétitifs, nous avons développé un système d'**Ajout en Masse**.

- **Interface** : Une fenêtre modale (Popup) charge dynamiquement le catalogue des compétences restantes
- **Mécanique** :
  - Utilisation de **ViewModels spécifiques** (`OffreBulkCompetenceViewModel`) contenant des listes d'objets avec état (`IsSelected`)

- Le recruteur coche plusieurs compétences, définit les niveaux pour chacune, et valide en une seule fois
- Le backend traite la liste et insère toutes les lignes dans la table de liaison (`CompetenceSouhaitee` ou `CompetenceAcquise`) en une transaction

6.4. Standardisation de l'Interface (NexaTheme UI)

L'interface a été uniformisée sur tous les modules (Offres, Candidats, Postes, Compétences) pour réduire la charge cognitive de l'utilisateur :

Layout :

- Barre de recherche et filtres compacts en haut
- Liste principale occupant 75% de la largeur (Col-9)
- Panneau latéral de contexte (Graphiques, Stats) occupant 25% (Col-3)

Composants Visuels

- Utilisation de **Cartes (Cards)** avec ombres douces (shadow-sm) pour délimiter le contenu
- Code couleur sémantique constant (Violet = Offres, Vert = Candidats, Jaune = Compétences)
- Graphiques **Chart.js** adaptés à l'espace (Barres horizontales pour les textes longs, Doughnut pour les répartitions)

---

7. Architecture Fonctionnelle Avancée

7.1. Problématique et Solution

Dans une architecture relationnelle classique, il est impossible de lier des entités enfants (ex: `CompetenceAcquise`) à une entité parent (ex: `Personne`) tant que celle-ci n'a pas été persistée en base de données (absence de Clé Primaire / ID).

Pour offrir une expérience utilisateur fluide sans bloquer la saisie, nous avons implémenté un **Pattern Wizard** (Assistant séquentiel) :

1. **Phase d'Initialisation (Create)** : L'utilisateur saisit les données scalaires (Nom, Ville...). Aucune compétence n'est demandée à ce stade

2. **Persistance Synchrone** : Le contrôleur sauvegarde l'entité et récupère immédiatement l'ID généré par SQL Server (SCOPE_IDENTITY)
3. **Phase d'Enrichissement (Edit)** : L'utilisateur est redirigé automatiquement vers l'écran d'édition, débloquant les fonctionnalités relationnelles (Ajout de compétences, Upload de documents)

7.2. Schéma du Flux de Données (Data Flow Diagram)

Ce schéma illustre le cycle de vie d'une création de profil candidat dans **NexaTalent**.

7.3. Intégrations API & Frontend

L'application ne fonctionne pas en vase clos. Elle s'appuie sur des services tiers pour garantir la qualité de la donnée.

Fonctionnalité	Technologie / API	Implémentation
Autocomplétion Ville	API Géo	Script JS écoutant l'événement input. Interroge l'API publique et remplit automatiquement les champs Ville et CodePostal
Cartographie	Leaflet.js + OSM	Affichage d'une carte interactive sur la fiche détail. Utilisation de **Nominatim** pour coder l'adresse textuelle en coordonnées GPS
Visualisation	Chart.js	Rendu des graphiques via injection de données JSON

8. Sécurité et Robustesse (Backend)

Pour garantir la fiabilité de l'application en production, plusieurs mécanismes de défense ont été implémentés dans le code C#.

8.1. Gestion des Conflits (Concurrency)

Lors de la modification d'une fiche (Edit POST), nous n'utilisons pas le mapping automatique risqué.

- **Méthode** : Nous récupérons l'objet original en base, et nous appliquons les modifications champ par champ
- **Avantage** : Cela évite l'écrasement accidentel de données non présentes dans le formulaire et prévient les erreurs de clé dupliquée (Duplicate Key Exception)

8.2. Validation des Données

- **Côté Client (jQuery Validate)** : Feedback immédiat (champ rouge) si un email est mal formaté ou un champ obligatoire manquant
- **Côté Serveur (Data Annotations)** : Validation stricte dans le modèle C# (`[Required]`, `[EmailAddress]`, `[Range]`) impossible à contourner
- **Logique Métier** : Vérification personnalisée (ex: interdiction d'une date de naissance dans le futur ou antérieure à 1900)

---

9. DIFFICULTÉS RENCONTRÉES ET SOLUTIONS

Pendant le développement, je me suis heurté à trois problèmes principaux qui m'ont obligé à revoir ma façon de coder.

9.1. Le problème de l'ID manquant (Création de profil)

- **Le problème :** Au début, je voulais créer un formulaire unique pour saisir le nom du candidat ET ses compétences en même temps. Mais j'ai eu une erreur bloquante : impossible d'enregistrer les compétences dans la base de données tant que le candidat n'a pas été créé (car il n'a pas encore d'ID).

La solution : J'ai compris que je devais séparer le processus en deux étapes.

1. D'abord, on crée la personne (SQL génère son ID).
2. Ensuite, on redirige automatiquement vers une page d'édition où l'on peut enfin ajouter les compétences grâce à l'ID qui existe maintenant.

9.2. Les lenteurs d'affichage

- **Le problème :** Quand j'ai testé ma liste avec 50 candidats, l'affichage est devenu très lent. En regardant de plus près, j'ai vu que mon code faisait une requête SQL pour récupérer la liste, puis une *nouvelle* requête pour *chaque* candidat afin de trouver ses compétences. Pour 50 candidats, je faisais 51 requêtes au lieu d'une seule.

- **La solution :** J'ai modifié ma requête Entity Framework en utilisant l'instruction `.Include()`. Cela force la base de données à tout renvoyer (Candidats + Compétences) en une seule grosse requête.

---

10. DIFFICULTÉS ET PROBLÈMES FONCTIONNELS

10.1. Incohérence des Données : Dissonance entre le statut professionnel et l'expérience

Une anomalie de logique métier a été relevée concernant la saisie des profils : le système autorise l'enregistrement d'un utilisateur déclarant un "Métier actuel" (indiquant une activité professionnelle active) tout en conservant un compteur d'années d'expérience à zéro. Cette contradiction sémantique nuit à la crédibilité de la base de données.

- **Préconisation :** Il est nécessaire d'implémenter une règle de validation croisée empêchant cette incohérence, ou idéalement, une logique de calcul automatique qui déduit l'expérience minimale en fonction de la date de début du poste actuel. Cela garantit qu'un professionnel en poste ne soit jamais techniquement considéré comme débutant absolu.

10.2. Logique Métier : Granularité de l'historique et règles de réembauche

Une carence a été identifiée dans la gestion du parcours candidat : le système ne distingue pas suffisamment l'expérience passée (archivée) de l'expérience actuelle, et l'exclusion pour réembauche se base uniquement sur l'intitulé du poste générique (ex: "Développeur Web"). Cette approche est trop restrictive car elle empêche un ancien collaborateur de postuler au même métier dans un contexte différent.

- **Préconisation :** L'architecture des données doit évoluer pour historiser non seulement le poste occupé, mais surtout l'**équipe de rattachement**. La règle de gestion doit passer d'une exclusion par *Poste* à une exclusion par *Contexte* :
  - **Interdiction :** Le système doit bloquer le couple **[Même Poste + Même Équipe précédente]** (impossibilité de reprendre exactement son ancien siège).
  - **Autorisation :** Le système doit permettre de postuler sur un intitulé de poste identique s'il est rattaché à une **nouvelle équipe** (mobilité transverse).

ANNEXE : IDENTIFIANTS PAR DÉFAUT & SÉCURITÉ

Dans le cadre du déploiement initial ou pour l'accès à l'environnement de démonstration, les comptes suivants ont été pré-configurés dans le script de `Seed` de la base de données (SQL Server).

Rôle	Email (Login)	Mot de passe (Temporaire)
Administrateur	admin@nexatalent.com	NexaAdmin2025!
Recruteur (User)	rh@nexatalent.com	Recruit#User01

> Documentation technique pour le projet NexaScore